Bridging Parallel and Reconfigurable Computing with Multilevel PGAS and SHMEM+

V. Aggarwal, A. George, K. Yalamanchili, C. Yoon, H. Lam, G. Stitt
NSF Center for High-Performance Reconfigurable Computing (CHREC)
ECE Department, University of Florida, Gainesville, FL 32611-6200
{aggarwal, george, yalamanchili, yoon, hlam, gstitt}@chrec.org

ABSTRACT

Reconfigurable computing (RC) systems based on FPGAs are becoming an increasingly attractive solution to building parallel systems of the future. Applications targeting such systems have demonstrated superior performance and reduced energy consumption versus their traditional counterparts based on microprocessors. However, most of such work has been limited to small system sizes. Unlike traditional HPC systems, lack of integrated, system-wide, parallel-programming models and languages presents a significant design challenge for creating applications targeting scalable, reconfigurable HPC systems. In this paper, we introduce and investigate a novel programming model based on Partitioned Global Address Space (PGAS), which simplifies development of parallel applications for such systems. The new *multilevel* PGAS programming model captures the unique characteristics of these systems, such as the existence of multiple levels of memory hierarchy and heterogeneous computation resources. To evaluate this multilevel PGAS model, we extend and adapt the SHMEM programming language to become what we call SHMEM+, the first known SHMEM library enabling coordination between FPGAs and CPUs in a reconfigurable, heterogeneous HPC system. Our design of SHMEM+ is highly portable and provides peak communication bandwidth comparable to vendor-proprietary versions of SHMEM. In addition, applications designed with SHMEM+ yield improved developer productivity compared to current methods of multi-device RC design and achieve a high degree of portability.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming – *parallel programming*.

General Terms

Design, Languages, Performance

Keywords

Reconfigurable computing, parallel programming, programming language, programming model, productivity, portability.

1. INTRODUCTION

High-performance computing (HPC) is a critical enabling technology for the advancement of science and engineering, supporting multi-scale simulations and experiments that drive breakthroughs in an ever-broadening range of fields. The field of HPC is currently undergoing a major transformation brought on by advances in device technologies as well as new generations of fixed-logic [1][2][3], reconfigurable-logic [4][5], and/or heterogeneous multicore and many-core [2][6][7] devices. These technologies are driving systems to become ever more powerful and efficient but unfortunately also more complex and demanding, with multiple types and levels of hardware parallelism to be understood and exploited.

A special class of such systems featuring reconfigurable computing, based on closely-coupled microprocessors and FPGAs, offers an attractive solution for HPC [8][9]. Numerous studies have demonstrated that such systems can achieve performance improvements ranging from 10× to more than 1000× over their microprocessor-based counterparts while concomitantly reducing energy consumption. Despite their superior performance, RC systems have failed to capture the HPC market, largely because of increased application-design complexity. Although advances in languages and tools for FPGAs have simplified device-level design for FPGAs, system-level design issues have largely been unaddressed. Such is the case with communication and synchronization between multiple devices in RC systems. Unlike traditional HPC systems, lack of integrated, system-wide, parallel-programming models and languages has limited most RC applications to small systems. The characteristic differences between RC systems and traditional HPC systems, such as additional levels of memory in the system and different execution models of heterogeneous devices present in the system, warrant a new programming model which can address these differences. Currently, application developers for large-scale RC systems employ ad-hoc methods and multiple libraries/APIs to incorporate inter- and intra-node communication and synchronization for such systems. As a result, the development productivity for scalable, parallel RC applications has suffered.

Although shared-memory models have been prevalent in HPC for decades, recently, newer models providing a programmer with a partitioned, global address space (PGAS) view for abstraction have been gaining popularity, such as Unified Parallel C (UPC) [10], SHMEM [11], Co-Array Fortran [12], and Titanium [13]. By extending the memory hierarchy to include an additional, higher-level global memory layer that is partitioned between nodes in the system, these languages/libraries allow for explicit or implicit one-sided data exchange (i.e. put, get) through reading and writing of global variables. Although designed for traditional HPC systems, these models have the requisite simplicity, syntax, and semantics to meet the needs of coordination amongst FPGA and CPU devices in a reconfigurable HPC system. However, these models need to be adapted and extended to work with such systems. For example, the concept of memory virtualization needs

HPRCTA'09, November 15, 2009, Portland, Oregon

to be extended to abstract the different levels of memory present in RC systems.

In particular, the SHMEM programming model stands out as a strong candidate for extending to RC systems due to its innate simplicity, low overhead, support for a partitioned global address space (PGAS), and emphasis upon explicit, fast one-sided communications. However, architectural differences introduced by incorporating RC devices in the system warrant re-examination of some concepts and semantics traditionally associated with its programming model. In this paper, we introduce a *multilevel* PGAS programming model for RC systems, which abstracts the details of multiple levels of memory available in the system, and presents the designer with a unified view of the system memory. Furthermore, we evaluate this model, through a prototype of SHMEM+ (i.e. an extended, adapted SHMEM library), the first known implementation of SHMEM that enables communication and synchronization between FPGAs and CPUs in a scalable, reconfigurable HPC system. Using SHMEM+, designers can create scalable, parallel applications that execute over a mix of microprocessors and FPGAs. The higher level of abstraction provided by SHMEM+ can yield significant improvement in developer productivity. Concomitantly, for the decomposed tasks of a parallel application, developers of FPGA cores can employ high-level synthesis tools and languages (e.g. Impulse-C, Carte-C, Handel-C, etc.) for creating hardware designs for FPGAs to further improve productivity. We analyze the performance of our prototype of SHMEM+ and investigate its inherent strengths through a case study.

The remainder of this paper is organized as follows. Section 2 describes some previous work. Section 3 provides a description of the *multilevel* PGAS programming model. Section 4 gives a detailed overview of the design of SHMEM+. In Section 5, experimental results are presented and the performance of different data-transfer routines available in SHMEM+ is analyzed. We also demonstrate a typical usage scenario with a case study of a content-based image retrieval application. Finally, Section 6 summarizes the work with conclusions and future work.

2. RELATED RESEARCH

Traditionally, developers of parallel programs have performed coordination between tasks using either message-passing libraries such as MPI [14] or shared-memory libraries such as OpenMP [15]. Recently, languages and libraries that present a partitioned global address space (PGAS) to the programmer, such as UPC [10] and SHMEM [11], have become more visible and popular. These languages provide a simple interface for developers of parallel applications through implicit or explicit one-sided data transfer functions, while providing comparable performance to message-passing libraries as demonstrated in [16]. However, since PGAS languages were developed for traditional HPC systems, they have been typically limited to homogeneous execution contexts of a cluster of microprocessors.

Owing to the emergence of a plethora of devices that are used for application acceleration and are coupled with microprocessors in HPC, there has been a quest for exploring parallel-programming models [17] that are better suited for such systems. Several research groups have recently shown interest in asynchronous execution in the PGAS model, leading to Asynchronous PGAS (APGAS), which lays the foundation for active-message programming and fine-grained concurrency [17]. APGAS also provides a framework for data transfer between non-coherent memories (DMA), remote atomic operations, and other techniques for overlapping computation and communication. APGAS is largely tailored towards spawning massively parallel, multi-threaded kernel computations at run-time, on accelerators such as GPUs, but inadequate for FPGAs where kernels are configured on the device as hardware engines, often at the beginning of the program.

Other researchers have attempted to build hybrid models using multiple models for a system [18]. System-level libraries and languages such as MPI and UPC were used for application coordination spanning nodes in clusters, and libraries such as OpenMP for coordination between tasks within each node. Hybrid models require the designer to partition their design into multiple levels and acquire expertise with multiple programming models and languages, libraries, and tools. By contrast, we attempt to abstract these details from the application designer and present an integrated programming model and library. In [19], the authors extend the UPC programming model to abstract a system of microprocessors and accelerators through a two-level hierarchy of parallelism. While their work shares the same goal of providing application developers with a unified programming model, their approach is quite different. In [19], the authors rely on identifying and extracting sections of code, specified in a UPC program, which are amenable to hardware acceleration and re-direct them through a source-to-source translator and a high-level synthesis tool to generate hardware designs. Instead of providing a means for creating hardware designs, we provide a parallel programming model, amenable to HPC systems which have a hierarchy of computational devices and memory resources, and leverage the efficiency of existing high-level synthesis tools to raise the abstraction for device-level design and generate the hardware.

Recently, much effort has been channeled towards standardizing the interface between microprocessors and accelerator devices such as FPGAs [20] and GPUs [21]. The focus of these efforts has been on the low-level interaction between the accelerator devices and x86 processors. The scope of our work is at a higher level and complements such efforts, as it can overlay a coordination framework on top of such APIs.

3. MULTILEVEL PGAS MODEL

Next-generation RC systems will be targeting FPGA devices in their system architectures in exotic ways to extract performance, ranging from closely coupled, in-socket accelerators to PCI-based accelerator cards. Figure 1 depicts an example RC system, where

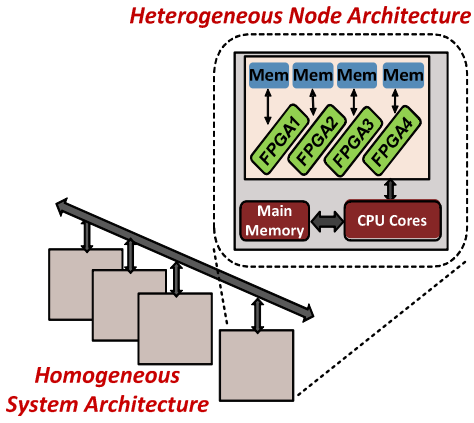


Figure 1. System architecture of a typical RC machine.

every node contains a set of processing units (PUs), each a microprocessor or FPGA. With FPGA devices and multicore CPUs, all within a single node, becoming pervasive in high-end computing systems, the existence of multiple levels of communication and memory hierarchy is becoming increasingly difficult to ignore. There is a need for a parallel-programming model that provides application designers with a higher level of abstraction, somewhat akin to that provided by the global memory layer in PGAS.

We introduce a *multilevel* PGAS model that integrates the different levels of memory hierarchy in the system, distributed within and across multiple nodes, into a partitioned, global address space. Figure 2 shows the physical distribution of memory components that form the global address space in the system. Memory blocks associated with all PUs in the system, irrespective of their physical location and hierarchy in the system architecture, can form a part of the virtual memory layer and have globally unique memory addresses in the system. Memory blocks that do not form a part of the global memory can be used by their PUs for storing local variables. It should be noted that memory blocks shown in Figure 2 correspond to the off-chip memory resources. On-chip memory structures of an FPGA such as block RAMs and register files are treated as local storage and not exposed as a part of PGAS. Such modeling of local storage is similar to that of microprocessor cache and registers, which are hidden from the PGAS layer in traditional HPC systems. Both CPUs and FPGAs provide interfaces required for the global memory abstraction for their corresponding memory blocks. In addition, each program instance of a SPMD (single-program, multiple-data) application in *multilevel* PGAS is comprised of multiple tasks, which are collectively executed by the PUs in a single node.

In systems equipped with multiple non-coherent memory blocks within a node, DMA operations are often employed for data transfer between different memory blocks, which are expensive operations and can significantly hamper application performance. *Multilevel* PGAS model requires data transfers between local memory components to be explicitly specified by the application designer. Having explicit calls for data transfers within a node gives more control to the designer and avoids the possibility of inefficiencies caused by transparent but expensive transfers which are implicitly embedded in the application code. In addition, *multilevel* PGAS model provides the designer with the ability of specifying affinity of various application data to specific memory components within a node during memory allocation. Therefore, the data can be placed in a memory block closer to the processing unit that operates on it most frequently. Both these features help in attaining higher performance for applications.

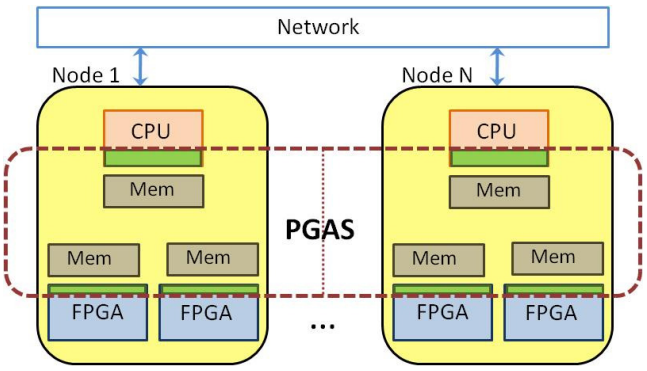
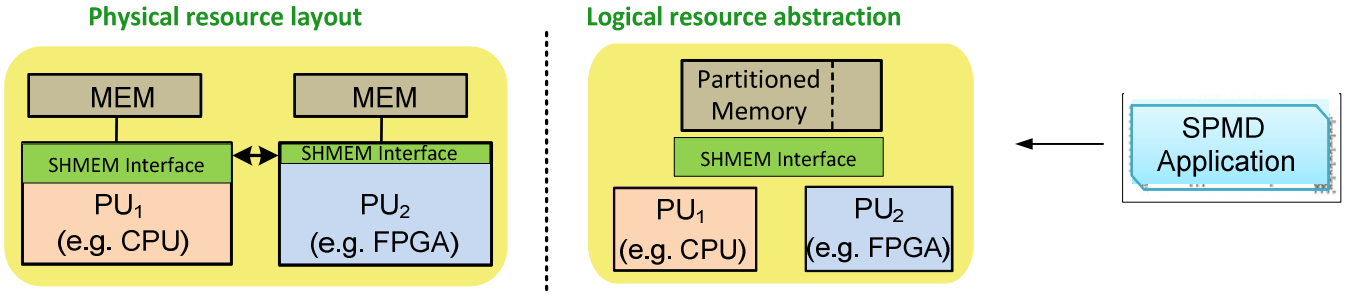


Figure 2. Distribution of memory and system resources in *multilevel* PGAS.

Figure 3 depicts the physical distribution of resources within each node and its equivalent logical abstraction provided by the *multilevel* PGAS model (used by SHMEM+). Although the figure depicts two processing units per node, one CPU and one FPGA, it can be generalized to include any number and variety. As shown in Figure 3(a), the global address space, partitioned across multiple nodes in the system, is composed of memory blocks which are physically distributed across different processing units within a node. However, the logical abstraction presented to a designer (shown in Figure 3(b)) integrates these physically separate memory blocks into a flattened view of the node's shared memory. Thus, application designers do not have to understand the distribution of data over the physical memory resources when accessing a remote node.

The PGAS interface, which allows the processing units to access different memory blocks in the system, is responsible for providing to application designers an abstraction of a single, integrated memory block. The physical implementation of the interface itself is system-dependent and can be realized in different ways by system architects. While each node provides the entire functionality required by the PGAS interface, each PU within a node is responsible for implementing only a subset of this functionality. The distribution of these responsibilities amongst the PUs within a node is dictated by their capabilities in the system. For example, in our initial study, the CPU provides a majority of the SHMEM functionality and the FPGA provides assistance for transfers to and from the FPGA's memory. As future work, we intend to investigate feasibility of FPGA-initiated transfers, which will require more extensive support from FPGAs.



(a) (b)

Figure 3. (a) Physical resource layout of a typical RC system, (b) Logical abstraction provided by multilevel PGAS model.

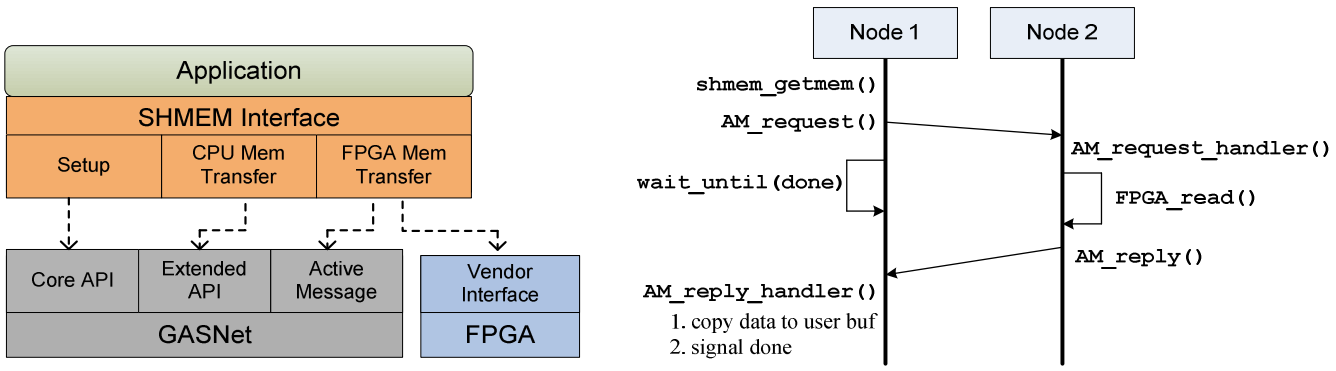


Figure 4. (a) Software architecture of SHMEM+, (b) Data transfer example using Active Messages.

4. DESIGN OVERVIEW OF SHMEM+

Based on this *multilevel* PGAS programming model, we extend the API of SHMEM to become what we call SHMEM+, which enables design of parallel applications for reconfigurable HPC systems. Using SHMEM+, designers can create highly scalable applications that execute over a mix of microprocessors and FPGAs. Previous implementations of the SHMEM API have targeted specific systems [11][22] and often lacked portability. SHMEM+ is built over services provided by Global Address Space NETworking (GASNet) from UC Berkeley [23]. GASNet is language-independent, communications middleware that provides network-independent, high-performance primitives tailored for implementing parallel GAS languages. As a result, SHMEM+ can be easily ported to systems that are supported by GASNet by simply modifying the FPGA interfaces that employ vendor-specific APIs.

Figure 4(a) shows the software architecture of SHMEM+. It makes use of GASNet's Core API, Extended API, and Active Message (AM) services. The setup functions, which perform memory allocation and other initialization tasks, employ the "Core API" services of GASNet. The data transfers to/from the CPU memory were built using the "Extended API," which provides direct support for high-level operations such as remote memory access. As a result, the SHMEM+ functions that perform transfers between two CPUs can be implemented by simply providing wrappers around the underlying GASNet functions. Since transfers to/from FPGA memory are not directly supported by underlying GASNet functions, they were developed using the AM service in conjunction with FPGA interfaces that we developed for our platform (more details about our experimental testbed are provided in Section 5). Figure 4(b) shows the sequence of steps involved in a transfer from FPGA memory using Active Messages. The message handlers shown in the figure employ FPGA_read/write functions, which we developed using the FPGA-board vendor's API to communicate with FPGA memory. Due to overhead incurred by AM services and data access to/from the FPGA board, communication with FPGA memory results in higher latency and lower bandwidth when compared to CPU memory transfers.

Our initial design of SHMEM+ as described in this paper focuses on a subset of baseline functions selected from the entire API function set of SHMEM. In this paper, we discuss 10 baseline functions shown in Table 1, which include five setup functions, four point-to-point messaging calls, and a collective synchronization routine. Some of these functions can be easily

Table 1. Baseline functions supported in prototype version of SHMEM+.

Function	SHMEM+ Call	Type	Purpose
Initialization	*shmem_init*	Setup	Initializes a process to use the SHMEM library
Communication Id	*my_pe*	Setup	Provides a unique node number for each process
Communication size	*num_pes*	Setup	Provides the number of PEs in the system
Finalize	*shmem_finalize*	Setup	De-allocates resources and gracefully terminates
Malloc	*shmalloc*	Setup	Allocates memory for shared variables
Get	*shmem_int_g*	P2P	Reads single element from the remote node's shared space
Put	*shmem_int_p*	P2P	Writes single element from the remote node's shared space
Get	*shmem_getmem*	P2P	Reads any contiguous data type from a remote PE
Put	*shmem_putmem*	P2P	Writes any contiguous data type to a remote PE
Barrier Synchronization	*shmem_barrier_all*	Collective	Synchronizes all the nodes together

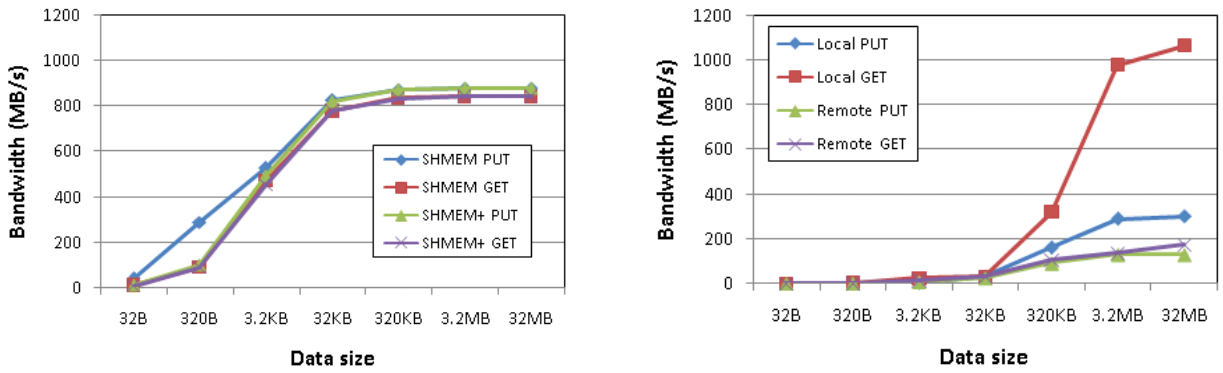


Figure 5. Bandwidth of point-to-point routines for transfers between (a) CPU and CPU, (b) CPU and FPGA.

extended to support other SHMEM functions; such is the case for single-element and contiguous data-transfer routines. In our initial version, we focus only on blocking and synchronous communications, which is the dominant mode in many applications. Blocking and synchronous communications require corresponding communication calls to complete data transfer before returning control to calling application. Therefore, our preliminary baseline does not include some other SHMEM functions which are commonly employed with asynchronous communication, such as fence, quiet, etc. These additional functions are left for future work.

It is our objective to keep the SHMEM+ interface consistent with previous SHMEM implementations. However, the functionality provided by SHMEM+ has been extended to incorporate support for FPGAs and provide *multilevel* PGAS abstraction. Thus, SHMEM+ functions perform these extra tasks in addition to the ones performed by traditional SHMEM routines. For example, the *shmem_init* call performs FPGA initialization (i.e. configuration of FPGA with the required hardware design) and FPGA memory-management operations, concomitant to CPU memory-segment initialization and management as performed by the traditional *shmem_init* function. The data-transfer functions (variations of shmem_get/put) perform exchanges between two CPUs or between a CPU and an FPGA. Based on the target memory address specified in the function, SHMEM+ identifies whether the target location resides in CPU or FPGA memory and adopts appropriate means of transferring the data. In addition, transfers to both remote and local FPGAs can be performed through the same interface, eliminating the need of multiple APIs. Without SHMEM+, application developers must orchestrate the transfers through multiple libraries based on the location and type of the target device. The memory allocation routine (*shmalloc*), which allocates memory for shared data variables from the shared address space, has been modified slightly to allow users to specify the affinity of any data to a particular memory block in the system.. For example, a set of data that is operated upon by an FPGA can be specified to be allocated on FPGA memory which, as explained in Section 3, can be beneficial for application performance. The application developer conveys this information by specifying the "type" parameter (type = 0 for CPU memory, 1 for FPGA memory) in the SHMEM+ function call.

5. EXPERIMENTAL RESULTS

Our experimental testbed consists of four server nodes connected via QsNet^{II} interconnect technology from Quadrics. Each node is a Linux server comprised of an AMD 2GHz Opteron 246 processor and equipped with a GiDEL PROCStar-III FPGA board. The FPGA board features four Altera Stratix-III EP3SE260 FPGAs, each with two external DDR memory modules of 2GB and one on-board 256MB DDR2 SDRAM bank. The FPGA board sits in a PCI-express ×8 slot. For our experiments and analysis, we currently employ only one FPGA per board, though our design can be easily extended to more FPGAs with minor modifications. In this section, we present the performance obtained for various memory transfers with SHMEM+ and compare it against the performance obtained with the vendor-proprietary, CPU-only version of SHMEM provided by Quadrics for QsNet systems.

5.1 Performance Analysis

Figure 5(a) shows performance of point-to-point communication routines in SHMEM+ for transfers between two CPUs. Bulk communication routines such as *shmem_getmem* and *shmem_putmem* attain a peak throughput of over 850MB/s for all such transfers. The bandwidth obtained with SHMEM+ calls, for transfers between two CPUs, is comparable to the proprietary version of SHMEM available from Quadrics. The SHMEM+ routines for these transfers benefit from direct support provided by GASNet and thus incur minimal overheads. The bandwidth is observed to saturate at approximately 890MB/s for data transfers larger than 320KB.

Figure 5(b) shows performance of data transfers between a CPU and an FPGA using SHMEM+ routines. The "Local PUT" and "Local GET" labels represent the bandwidth of data transfers between a host CPU and its local FPGA on the same node. The bandwidth of such local transfers is specific to the particular FPGA board and depends upon a variety of factors associated with interconnect(s) between CPU and FPGA, efficiency of the communication controller on the board, etc. Most RC systems offer a higher bandwidth for read operation from an FPGA (FPGA to CPU) when compared to write operation (CPU to FPGA). Similarly, our system yields a peak bandwidth of over 300MB/s for local put operations (CPU to FPGA) and approximately 1000MB/s for local get operations (FPGA to CPU). The "Remote PUT" and "Remote GET" labels in Figure 5(b) represent the bandwidth of data transfers between a CPU and an FPGA on different node. As expected, the bandwidth for such transfers is observed to be lower than the bandwidth attained for local transfers; peak bandwidth observed is under 200MB/s. These transfers were implemented using GASNet's medium AM service, which enforces a maximum packet size of 63KB and causes

substantial degradation in performance. An additional performance penalty is incurred while transferring these small data packets to the FPGA memory over PCIe. From the results presented in this section, it can be observed that SHMEM+ performs well for transfers between two CPUs and reasonably well between a CPU and an FPGA.

5.2 Case Study

To exemplify the programming philosophy of SHMEM+, we present the design of a Content-Based Image Retrieval (CBIR) application. CBIR is a common application in computer vision and consists of searching a large database of digital images for the ones that are visually similar to a given query image, where the search is based on contents of the image. The content in this context can be one of the several features present in the image, such as colors, shapes, textures, or any other information that can be derived from the image. CBIR has been widely adopted in many domains such as biomedicine, military, commerce, education, and Web image classification and searching. Each image in a CBIR system is represented by a *feature vector*, which is based on characteristics of the image as cited above. Similarity between a query image and the set of images in the database is determined by measuring similarity between their feature vectors. The processes of determining the feature vector and analyzing images for similarities are often the most computationally intensive stages in any CBIR system [24]. There are various forms of parallelism available in the application that can be exploited by RC systems to accelerate the search process [25].

Our implementation presented here employs a technique based on auto-correlogram of color components [26], where the feature vector is based on color information in the image. A correlogram of an image corresponds to a table where the rows are indexed by color pairs (c_i, c_j) such that the d-th column in row (c_i, c_j) stores the probability of finding a pixel of color c_j at a distance d from a pixel of color c_i in the image. For the case of auto-correlogram, the table only consists of rows where $c_i = c_j$. In this paper, we use a modified version of auto-correlogram, which employs an absolute count of the occurrences of a pixel of color c_i instead of the probability of such an event. Similarity between two images is determined by calculating the Sum of Absolute Differences (SAD) between their feature vectors.

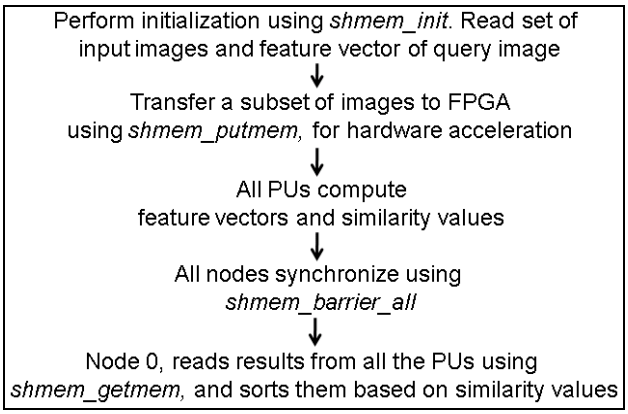


Figure 6. Processing steps involved in our parallel algorithm for CBIR using SHMEM+

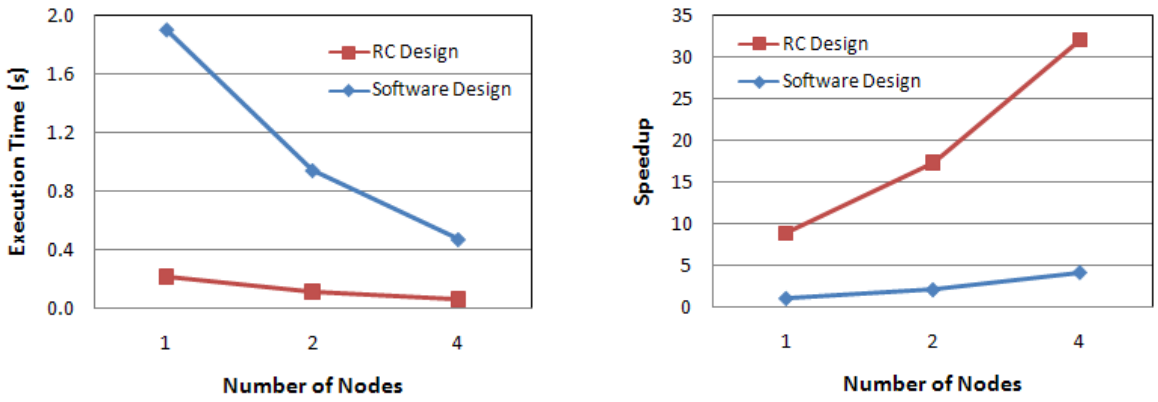
The basic parallel algorithm employed in our experiments distributes the set of images to be searched over a set of nodes and allows multiple processing units to evaluate these images simultaneously, as shown in Figure 6. The steps involved in our parallel implementation are as follows:

1. All nodes perform initialization using *shmem_init*, which also configures the FPGA with the desired bitfile (specified as a command line parameter).
2. CPUs on all nodes read their subset of input images from a storage device (such as a local hard disk or a network storage device) along with the feature vector of the query image.
3. CPUs then each transfer a subset of the set of images to local FPGA memory using the *shmem_putmem* function, for processing in reconfigurable hardware.
4. CPUs initiate the execution on their local FPGAs through a "GO" signal. CPUs and FPGAs on all nodes compute feature vectors and similarity measures for their subset of images in parallel.
5. FPGAs signal the completion of execution to local CPUs through a "DONE" signal. Once computation on CPUs and FPGAs on each node is complete, all nodes synchronize using *shmem_barrier_all*.
6. Finally, Node 0 reads similarity values from all CPUs and FPGAs using the *shmem_getmem* function. Results are then sorted in decreasing order of similarity.

In addition to software parallelism described in the algorithm above, hardware designs for FPGAs in our system instantiate multiple computational kernels and operate on five images in parallel. Figure 7 shows the execution times and the speedup versus a serial software baseline for different system sizes. Our experiments were conducted for an image size of 128×128, with the search database consisting of 800 images. Since the sorting process involved in Step 5 is the same for the serial baseline and forms an insignificant part of the execution time, it was omitted from the execution times recorded in our experiments.

A linear speedup is obtained for the parallel software designs, as each node now operates over a subset of the set of images. When RC components on each node are employed in addition to microprocessors, over 30× speedup is obtained with four nodes. The FPGAs were able to process images at a much faster rate than the CPUs leading to significant improvement in application performance. As SHMEM+ is further optimized for PCIe and even faster intra-server interconnects, such as HyperTransport (HT) and Quick Path Interconnect (QPI), the performance of SHMEM+ applications is likely to improve even further.

More importantly, SHMEM+ provides application developers with a parallel-programming model that enables productive and portable design of scalable RC applications, as demonstrated through the case study in this section. SHMEM+ simplifies the design process for such applications by raising the level of abstraction, somewhat akin to methods employed for traditional parallel applications. Without SHMEM+, application developers must employ multiple libraries with varying APIs to incorporate communication amongst a cluster of host CPUs and to facilitate coordination between host CPUs and FPGAs. In addition, any communication with an FPGA on a remote node will have to be explicitly routed by the developer, through the host CPU on that remote node. With SHMEM+, the developer is oblivious to such details and exposed to a higher level of abstraction, which can lead to significant improvement in developer productivity. Since



(a) (b)

Figure 7. Performance comparison of parallel CBIR application designed with SHMEM+. Software designs involve only CPU devices whereas RC designs involve both CPUs and FPGAs on each node. (a) Execution time of different designs, (b) Speedup versus a serial software baseline for different designs.

SHMEM+ applications do not employ any vendor-specific APIs for interaction with FPGAs, applications are highly portable. As long as the SHMEM+ library can be supported on a target RC system, any application designed with SHMEM+ can execute on it without requiring changes to the application source code.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we presented a parallel-programming model and a communication library for scalable, heterogeneous, reconfigurable systems. The *multilevel* PGAS model introduced in this paper is able to capture key characteristics of RC systems, such as different levels of memory hierarchy and differences in the execution model of heterogeneous devices present in the system. The existence of such a programming model will enable development of scalable, parallel applications for reconfigurable HPC systems.

Based on the *multilevel* PGAS programming model, we presented the design of SHMEM+ (an extended SHMEM library), the first known version of SHMEM that enables designers to create scalable applications that execute over a mix of microprocessors and FPGAs. Initial results from experiments demonstrate that transfers between two CPUs achieve high bandwidth, comparable to the vendor-proprietary version of SHMEM on the same system. Transfers to FPGA memory incur extra penalties, and therefore achieve lower peak bandwidth. Our case study demonstrated the simplified design process involved with SHMEM+ for developing scalable RC applications. The design steps involved in developing applications with SHMEM+ are very similar to traditional methods for development of parallel applications. CBIR application developed using SHMEM+ achieved a speedup of over 30× when the RC components on each node were employed in addition to the microprocessors. More importantly, the higher level of abstraction provided by SHMEM+ to the application developer leads to significant improvement in productivity. In addition, by hiding details of vendor-specific FPGA communication from the user, SHMEM+ creates highly portable applications.

For future work on portability and scalability studies, we plan to port and evaluate SHMEM+ on our new Novo-G machine, an RC system comprised of twenty-four compute nodes, each equipped with four large, top-of-the-line FPGAs and a quad-core microprocessor. Also, we plan to enhance the communication model by investigating mechanisms for FPGA-initiated transfers. In addition, we intend to investigate, develop, and evaluate tools to support performance analysis for SHMEM+ applications.

7. ACKNOWLEDGMENTS

This work was supported in part by the I/UCRC Program of the National Science Foundation under Grant No. EEC-0642422. The authors gratefully acknowledge vendor equipment and/or tools provided by GiDEL that helped make this work possible. The authors also thank Rafael Garcia, M.S. student, in our lab for his contributions to this work.

8. REFERENCES

[1] Ambric, Inc. 2008. Ambric technology backgrounder. http://www.ambric.com/technology/technology-overview.php.

[2] F- Chen, T., Raghavan, R., Dale, J. N., and Iwata, E. 2007. Cell broadband engine architecture and its first implementation: a performance view. IBM Journal of Research and Development 51, 5, 559-572.

[3] ClearSpeed Technology PLC. 2007. CSX600 Architecture. Whitepaper. ClearSpeed Technology PLC.

[4] Altera Corp. 2008. Stratix IV Device Handbook. Altera Corp.

[5] Xilinx, Inc. 2008. Virtex-5 Family Overview. Xilinx, Inc.

[6] IBM Corp.2008. PowerXCell 8i Processor Specifications. IBM Corp.

[7] Tilera Corp. 2008. TILE64 Processor Product Brief. Tilera Corp.

[8] SRC Computers, Inc. 2009. MAPstation workstations. www.srccomp.com/products/mapstation.asp, (accessed on July 12, 2009).

[9] XtremeData, Inc. 2009. In-Socket Accelerators. http://www.xtremedatainc.com/index.php?option=com_content&view=article&id=109&Itemid=170, (accessed on July 12, 2009).

[10] Carlson, W. W., Draper, J. M., Culler, D. E., Yelick, K., Brooks, E., and Warren, K. 1999. Introduction to UPC and language specification. University of California-Berkeley. Technical Report: CCS-TR-99-157, 1999.

[11] SHMEM website. http://docs.cray.com/books/004-2518-002/html-004-2518-002/z826920364 dep.html, (accessed on July 12, 2009).

[12] Numrich, B. and Reid, J. 1998. Co-Array Fortran for Parallel Programming. ACM Fortran Forum, 17, 2, (1998), pp. 1–31.

[13] Yelick, K., Semenzato, L., Pike, G., Miyamoto, C., Liblit, B., Krishnamurthy, A., Hilfinger, P., Graham, S., Gay, D., Colella, P., and Aiken, A. 1998. Titanium: A High-Performance Java Dialect. Workshop on Java for High-Performance Network Computing (June 1998), Las Vegas, Nevada.

[14] MPI website. http://www.mcs.anl.gov/research/projects/mpi/, (accessed on July 12, 2009).

[15] Open MP website. http://openmp.org/wp/, (accessed on July 12, 2009).

[16] Nishtala, R., Hargrove, P.H., Bonachea, D.O. and Yelick, K.A. 2009. Scaling communication-intensive applications on BlueGene/P using one-sided communication and overlap. IEEE International Parallel & Distributed Processing Symposium (May 23-29, 2009). pp.1-12.

[17] Workshop on Asynchrony in the PGAS Programming Model. http://research.ihost.com/apgas09/, (accessed on July 12, 2009).

[18] Farreras, M., Marjanovic, V., Ayguade, E., and Labarta, J. 2009. Gaining asynchrony by using hybrid UPC/SMPSs. Workshop on Asynchrony in the PGAS Programming Model (June 2009), Yorktown Heights, NY.

[19] El-Ghazawi, T., Serres, O., Bahra, S., Huang, M. and El-Araby, E. 2008. Parallel Programming of High-Performance Reconfigurable Computing Systems with Unified Parallel C. Proc. of Reconfigurable Systems Summer Institute (July 7-9, 2008) RSSI 2008. Urbana, Illinois.

[20] OpenFPGA GenAPI version 0.4 Draft For Comment. 2009 http://www.openfpga.org/pages/Standards.aspx, (accessed July 12, 2009).

[21] OpenCL 1.0 Specification. 2009. http://www.khronos.org/registry/cl/specs/opencl-1.0.43.pd, (accessed July 12, 2009).

[22] Intro_shmem - Introduction to the SHMEM programming model. http://docs.sgi.com/library/tpl/ cgi-bin/getdoc.cgi?coll=linux&db=man&fname=/usr/share/catman/man3/intro_shmem.3.html&srch= intro_shmem, (website accessed July 12, 2009).

[23] Bonachea, D. and Jeong, J. 2002. GASNet: A Portable High-Performance Communication Layer for Global Address-Space Languages. CS258 Parallel Computer Architecture Project, Spring 2002.

[24] Guyon, I., Gunn, S., Nikravesh, M., and Zadeh, L. 2006. Feature Extraction, Foundations and Applications. Springer, 2006.

[25] Skarpathiotis, C. and Dimond, K. R. 2004. Hardware Implementation of Content Based Image Retrieval Algorithm. Springer LNCS 3203, pp. 1165-1167.

[26] Huang, J., Kumar, S.R., Mitra, M., and Zhu, W.J. 1997. Image indexing using color correlograms. Proc. of IEEE Computer Society Conference on Computer Vision and Pattern Recognition (1997). San Juan, Puerto Rico. pp. 762-768.

[27] Huang, J., Kumar, S.R., Mitra, M., and Zhu, W.J. 1998. Spatial color indexing and applications. Proc. of sixth International Conference on Computer Vision(1998), Bombay, India. pp. 602-607.

[28] Ojala, T., Rautiainen, M., Matinmikko, E. and Aittola, M. 2001. Semantic Image Retrieval with HSV correlograms. Proc. of twelfth Scandinavian Conference on Image Analysis (2001). Bergen, Norway. pp. 621-627.